EXHIBIT 2

Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks [*]

Elena Gabriela Barrantes
University of New Mexico
gbarrant@cs.unm.edu

David H. Ackley
University of New Mexico
ackley@cs.unm.edu

Stephanie Forrest
University of New Mexico
forrest@cs.unm.edu

Trek S. Palmer [†]
University of New Mexico
tpalmer@cs.unm.edu

Darko Stefanović
University of New Mexico
darko@cs.unm.edu

Dino Dai Zovi [‡]
University of New Mexico
ddaizovi@atstake.com

ABSTRACT

Binary code injection into an executing program is a common form of attack. Most current defenses against this form of attack use a ‘guard all doors’ strategy, trying to block the avenues by which execution can be diverted. We describe a complementary method of protection, which disrupts foreign code execution regardless of how the code is injected. A unique and private machine instruction set for each executing program would make it difficult for an outsider to design binary attack code against that program and impossible to use the same binary attack code against multiple machines. As a proof of concept, we describe a *randomized instruction set emulator* (RISE), based on the open-source Valgrind x86-to-x86 binary translator. The prototype disrupts binary code injection attacks against a program without requiring its recompilation, linking, or access to source code. The paper describes the RISE implementation and its limitations, gives evidence demonstrating that RISE defeats common attacks, considers how the dense x86 instruction set affects the method, and discusses potential extensions of the idea.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection; D.3.4 [**Programming Languages**]: Processors

General Terms

Security, Languages

[*]This paper will appear in the 10th 10th ACM Conference on Computer and Communications Security (CCS), Oct. 27-31, Washington DC, USA, 2003

[†]Currently at the University of Massachusetts, Amherst.

[‡]Currently at @stake Inc.

Keywords

Automated Diversity, Security, Emulation, Language Randomization, Obfuscation, Information Hiding

1. INTRODUCTION

Standardized interfaces between software and hardware are a double-edged sword. On the one hand, they lead to huge productivity improvements through independent development and optimization of hardware and software. But, they also allow a single attack code designed against an exploitable flaw to gain control of thousands or millions of standardized systems. One approach to controlling this form of attack is to ‘destandardize’ the protected system in an externally unobservable way, so that an outside attacker either cannot easily obtain the information needed to craft the attack or must manually regenerate the attack once for each new attack instance. Techniques that take this approach include obfuscation, information hiding, and automated diversity.

In the case of binary code injection, many defense techniques act to block known routes by which foreign code is placed into the execution path of a program. For example, stack defense mechanisms [20, 37] that protect return addresses defeat large classes of buffer overflow attacks; separate techniques [18, 32] defeat buffer overflows in other write-accessible parts of address space. Attacks such as ‘return into libc’ [30] avoid injecting any executable code at all, instead altering only data and addresses so that code already existing in the program is subverted to execute the attack; defense techniques like address obfuscation [16, 14, 32] counter by hiding and/or randomizing existing code locations.

In addition to such ‘perimeter defense’ techniques aimed at specific attack vectors, a secret destandardization of the executing code itself offers a complementary and quite general method of protection. With such instruction set obfuscation, each program (or process, or machine, or other unit of machine code execution) has a different and secret instruction set. If the number of possible instruction sets is large and externally unobservable, the cost of developing an attack from the outside is greatly increased, and different attacks must be crafted for each protected system.

In this paper we describe *randomized instruction set emulation* (RISE), an instruction set obfuscation technique implemented at the machine emulator level. Each byte of protected code in the program is individually scrambled using pseudorandom numbers seeded with a random key that is unique to each program execution. With the scrambling constants it is trivial to transform the obfuscated code back to normal instructions executable on the physical machine, but without knowledge of the key it is infeasible to produce even a short code sequence that implements any given behavior. Foreign binary code that reaches the path of execution will be descrambled without ever having been correctly scrambled, producing essentially random bits that will usually crash the program under attack.

1.1 Threat model

RISE does not address, let alone solve, all possible security problems or even all possible attacks via communications networks. Our specific threat model is *binary code injection from the network into an executing program*. This includes many real world attacks, but explicitly excludes others, such as macro viruses that involve injection of something other than binary code, or the 'data injection' attacks mentioned above that do not rely on machine code. We assume that attacks arrive via network communications and that the contents of local disks are therefore trustworthy before an attack has occurred.

Our threat model is related to, but distinct from, other models used to characterize buffer overflow attacks [21, 18], so it is important to compare and contrast the approaches. Our threat model includes any attack in which native code is injected into a running binary, including misallocated malloc headers, footer tags [2], and format string attacks that can write a byte to arbitrary memory locations without actually overflowing a buffer [31]. RISE will protect against injected code arriving by any of these methods. On the other hand, other buffer overflow defenses, such as the address obfuscation mentioned earlier, can prevent attacks that are specifically excluded from our code-based threat model. RISE provides no defense against data-only attacks, which can range from the modification of jump addresses and parameters to call an existing library function (such as the family of return-into-libc attacks [30]) to the modification of password files or other critical information (for example, a privilege escalation as in [4]).

We envision the relatively general code-based mechanism of RISE being used in conjunction with more specific data- and address-based mechanisms to provide deeper, more principled, and more robust defenses against both known and unknown attacks.

1.2 Overview

In this paper we present a proof-of-concept RISE system, building randomized instruction set support into a version of the Valgrind x86-to-x86 binary translator [36]. In Section 2 we describe a *randomizing loader* for Valgrind that scrambles code sequences loaded into emulator memory from the local disk using a hidden random key. Then, during Valgrind's emulated instruction fetch cycle, fetched instructions are unscrambled, yielding unaltered x86 machine code runnable on the physical machine. The RISE design makes few demands on the supporting emulator and could be easily ported to any binary-to-binary translator for which source code is available.

In Section 3 we present our experimental results. We have found that RISE is successful in preventing code injection attacks, both synthetic and real, as described in Section 3.1. Section 3.2 analyzes the potential problem of creating valid instructions with the randomization given the dense x86 instruction set and Section 3.3 comments on performance issues.

When binary attack code, arriving over the network, exploits a bug and manages to interpose itself into the emulator execution path, the injected code will not have been scrambled by the loader. Consequently, when the attack code is fetched and unscrambled by the emulated instruction unit, it will appear as an essentially random string of bits. Despite the density of the x86 instruction set, we present data suggesting that the vast majority of random code sequences will encounter an address fault or illegal instruction quickly, aborting the program. Thus with RISE, an attack that would otherwise take control of a program is downgraded into a denial-of-service attack against the exploitable program. Regardless of what flaw is exploited in a protected program—whether well-known or entirely novel—the network binary code injection attack will fail with very high probability.

Section 4 summarizes related work, and Section 5 concludes with a general discussion.

2. TECHNICAL APPROACH AND IMPLEMENTATION

This section describes the prototype implementation of RISE using Valgrind [36] for the Intel x86 architecture. The RISE strategy is to provide each program copy its own unique and private instruction set. To do this, we consider what is the most appropriate machine abstraction level, how to scramble and descramble instructions, when to apply the randomization and when to descramble, and how to protect interpreter data. We also describe idiosyncrasies of Valgrind that affected the implementation.

2.1 Machine abstraction level

The native instruction set of a machine is a promising computational level for automated diversification. Since all computer functionality can be expressed in machine code, it is a desirable level to attack and protect. Also, with a network-based threat model, all legitimately executing machine code comes from the local disks, providing a clear trust boundary. By contrast, a Javascript interpreter in a web browser would be a poor candidate for this approach, because most Javascript code arrives over the network without firm trust boundaries between more and less legitimate code sequences.

A drawback of native instruction sets is that they are traditionally physically encoded and not readily modifiable. RISE therefore works at an intermediate level, using software that performs binary-to-binary code translation. The performance impact of such tools can be minimal [11, 15]. Indeed, binary-to-binary translators sometimes improve performance compared to running the programs directly on the native hardware [11]. For ease of research and dissemination, we selected an open-source system, Valgrind [36], as the basis for our demonstration implementation.

Although Valgrind is billed primarily as a tool for detecting memory leaks and other program errors, it contains a complete x86-to-x86 binary translator. The primary drawback of Valgrind is that it is very slow, largely due to its extensive access checking. However, the additional slowdown imposed by adding RISE to Valgrind is modest (see Section 3), and we are optimistic that porting RISE to a more performance-oriented emulator will yield a fully practical code defense.

2.2 Instruction set randomization

Instruction set randomization could be as radical as developing a new set of opcodes, instruction layouts, and a key-based toolchain capable of generating the randomized binary code. And, it could take place at many points in the compilation-to-execution spectrum. Although performing randomization early could help distinguish code from data, it would require a full compilation environment on every machine, and recompiled randomized programs would likely have one fixed key indefinitely. RISE randomizes as late as possible in the process, scrambling each byte of the trusted code as it is loaded into the emulator, and then unscrambling it before execution by the virtual machine. Deferring the randomization to load time makes it possible to scramble and load existing files in the Executable and Linking Format (ELF)[38] directly, without recompilation or source code, provided we can reliably distinguish code from data in the ELF file format.

The unscrambling process needs to be fast, and the scrambling process must be as hard as possible for an outsider to deduce. Our current approach is to generate at load time a pseudo-random sequence the length of the overall program text using the Linux `/dev/urandom` device [39], which uses a secret pool of true randomness to seed a pseudo-random stream generated by feedback through SHA1 hashing. The resulting bytes are simply XORed with the instruction bytes to scramble and unscramble them. If the underlying truly random key is long enough, and as long as it is infeasible to invert SHA1 [35], then we can have confidence that an attacker could not break the entire sequence. We return to the issue of how secure the RISE encoding in Section 5.

2.3 Design decisions

Two important aspects of the RISE implementation are how it handles shared libraries and how it protects the plaintext executable.

Much of the code executed by modern programs resides in shared libraries. This form of code sharing can significantly reduce the effect of the diversification, as processes must use the same instruction set as the libraries they require. When our load-time randomization mechanism writes to memory that belongs to shared objects, the Operating System does a copy-on-write, and a private copy of the scrambled code is stored in the virtual memory of the process. This significantly increases memory requirements, but increases interprocess diversity and avoids having the plaintext code mapped in the protected processes' memory.

Protecting the plaintext instructions inside Valgrind is a second concern. As Valgrind simulates the operation of the CPU, during the fetch cycle when the next byte(s) are read from program memory, RISE intercepts the bytes and unscrambles them; the scrambled code in memory is never modified. Eventually, however, a plaintext piece of the program (semantically equivalent to a basic block) is written to Valgrind's cache. From a security point of view, it would be best to separate the RISE address space completely from the protected program address space, so that the plaintext is inaccessible from the vulnerable program, but as a practical matter this would slow down emulator data accesses to an extreme and unacceptable degree. For efficiency, the RISE interpreter is best located in the same address space as the target binary, but of course this introduces some security concerns. A RISE-aware attacker could aim to inject code into a RISE data area, rather than that of the vulnerable process. This is a problem because the cache cannot be encrypted. To protect it, cache pages are kept as read and execute only. When a new translated block is ready to be written to the cache, we mark the affected pages as writable, execute the write action, and return them to their original non-writable permissions.

2.4 Implementation issues

An emulator needs to create a clear boundary between itself and the process to be emulated. In particular, the emulator should not use the same shared libraries as the process being emulated. Valgrind deals with this issue by adding its own implementation of any library function it requires using a local name, for example, `VGplain_printf(...)` instead of `printf(...)`. However, we discovered that Valgrind occasionally jumped into the target binary to execute low-level functions (e.g., `__umoddi` and `__udivdi`). When that happened, the processor attempted to execute instructions that had been scrambled for the emulated process, causing Valgrind to abort. Although this was irritating, it did demonstrate the robustness of the RISE approach in that these latent 'boundary crossings' were immediately detected. We worked around these dangling unresolved references by adding more local functions and renaming affected symbols with local names (e.g., rise_umoddi(...) instead of '%' (the modulo operator)).

A more subtle problem arises because the IA32 does not impose any data and code separation requirement, and some libraries still use dispatch tables stored directly in the code. In those cases the addresses in one of these internal tables are scrambled at load time (because they are in a code section), but are not descrambled at execution time because they are read as data. Although this does not cause an illegal operation, it causes the emulated code to jump to a random address and fail inappropriately. We solved this problem by adding machine code to check for internal references in the block written to the cache. If the reference was internal, we performed an additional descrambling operation on the address recovered as data.

An additional difficulty was discovered with Valgrind itself. For somewhat subtle reasons involving dynamic libraries, Valgrind has to emulate itself at certain moments, and it has a special workaround in its code to execute certain functions natively, avoiding an infinite emulation regress. We handled this by detecting Valgrind's own address ranges and treating them as special cases. We believe this issue is specific to Valgrind, and we expect not to have it in other emulators.

Attack	**Linux Distribution**	**Stopped by unmodified Valgrind**	**Stopped by RISE**
Synthetic heap overflow	N/A		√
Synthetic stack overflow	N/A		√
Apache OpenSSL SSLv2	RedHat 7.0 and 7.2	√	√
Apache mod php	RedHat 7.2	√	√
Bind NXT	RedHat 6.2		√
Bind TSIG	RedHat 6.2	√	√
CVS pserver double free	RedHat 7.3	√	√
SAMBA nttrans	RedHat 7.2	√	√
SAMBA trans2	RedHat 7.2		√
SSH integer overflow	Mandrake 7.2	√	√
rpc.statd format string	RedHat 6.2		√
sendmail crackaddr buffer overflow	RedHat 7.3	√	√
wuftpd format string	RedHat 6.2 to 7.3	√	√
wuftpd glob " {"	RedHat 6.2	√	√

Table 1: Results of attacks executed under Valgrind (without RISE) and RISE.

3. EXPERIMENTAL RESULTS

The results reported in this section were obtained using the RISE prototype, available under the GPL from http://cs.unm.edu/~immsec. We have tested RISE's ability to run programs successfully under normal conditions and its ability to disrupt a variety of machine code injection attacks (Section 3.1). In addition, we have tested the safety of executing instruction sequences after they have been randomized (Section 3.2) and concluded that programs randomized under RISE can execute with very low probability of doing damage. Finally, we make some observations about the performance of RISE (Section 3.3), concluding that the approach could be used in a production system if ported to a more efficient emulator.

3.1 Attacks

We tested two synthetic and a dozen real attacks. The synthetic attacks, published in [23], create a vulnerable buffer—in one case on the heap and in the other case on the stack—and inject shellcode into it. Without RISE, both attacks successfully spawned a shell, and with RISE, the attacks were stopped. The real attacks were launched from the CORE Impact attack toolkit [1]. We selected twelve attacks that satisfied the following requirements of our threat model and the chosen emulation tool: the attack is launched from a remote site; the attack injects binary code at some point in the execution; the attack succeeds on a Linux OS. Valgrind is specifically designed to run under Linux, and we tested several different Linux distributions, reporting data from two (RedHat from 6.2 to 7.3 and Mandrake 7.2).

All of the attacks were tested to make sure they were successful in the vulnerable application before retesting with RISE. The attacks were all successfully defeated by RISE (column 4 of Table 1). When we analyzed the logs generated by RISE, however, we discovered that 9 of the 14 tested attacks failed without ever executing the injected attack code (column 3). This class of attacks is notoriously fragile, and the mere fact of emulation can often disrupt them; one could imagine modifying the attacks to overcome the perturbations of the emulator, and in the future we hope to test these modified attacks against RISE.

The synthetic attacks and the more robust real attacks (Bind NXT, Samba trans2, and rpc.statd), were unaffected by the emulator's presence and all managed to establish a shell successfully when the target program was run on an unmodified version of Valgrind. However, all of them were stopped by RISE. Bind NXT and Samba trans2 attacks are both based on stack overflows, while the rpc.statd attack injects binary code into the GOT table.

These results confirm that we successfully implemented RISE and that a randomized instruction set prevents injected machine code from executing without the need for any knowledge about how or where the code was inserted in process space.

3.2 How safe is it to execute random instructions?

Defenses such as RISE depend on randomization to prevent an attacker from knowing precisely what an attack will do. If foreign machine code is injected into a RISE protected program without scrambling, then when it is unscrambled for execution it will be mapped to essentially random bytes and will not perform any specific function.

If such random code does not behave as intended, what does it do? The expectation is that random code strings will cause the attacked program to crash quickly, but we don't know a priori what will happen. The RISE prototype produces randomized instruction sets that are in byte-for-byte correspondence with actual x86 instructions, so the transformation process does not affect code size or layout. This avoids complexity and allows us to defer randomization until load time. But, with so much of the x86 opcode space already defined, there is a significant chance that a randomly scrambled opcode will be something other than an illegal instruction.

To test the safety of random instructions, we performed the following test: We built a small program that contained a rootshell exploit coded in x86 machine code (the shellcode from '`test-sc2.c`' in [6]). When the program ran, it first randomized the exploit code in place using a random number seed supplied on the command line. It then 'returned into' the randomized attack code following the pattern that could happen in an attack. We ran $30,000$ tests varying only the random seed, running the program under gdb to capture information about if, where, and why the program dies. Table 2 and Figure 1 summarize the results. Over 99.8% of randomizations lead to the program aborting by one of four signals. SIGILL is an illegal instruction, SIGFPE is a floating point exception (such as division by zero), and SIGSEGV and SIGBUS are two varieties of addressing problems. In the remaining cases, the program entered an (apparently) infinite loop. In none of the 30,000 test cases did the attack code manage to access the command interpreter `/bin/sh` as intended by the attacker.

Outcome		*Count*	*Percent*	*Cumulative*
Signalled		29,945	99.82%	99.82%
SIGSEGV	25,162			
SIGILL	4,504			
SIGFPE	178			
SIGBUS	101			
Looped(timeout)		55	0.18%	100.0%
Acquired shell		0	0.0%	0.0%
Total tests		30000		

Table 2: Outcomes of executing randomized shell acquisition code.

There are caveats to this data. Note that SIGSEGVs are by far the most commonly emitted signal—but that could be misleading because the test program is so small. A larger program with a correspondingly larger space of legal addresses would be expected to generate fewer SEGV's and more jumps to random but legal addresses, causing more complex and possibly subtly harmful behavior patterns.

Nonetheless, this case study suggests that the vast majority of randomizations of a genuine attack do indeed simply cause a program crash. Although this test does not directly answer the question of *how fast* the crashes tend to occur, Figure 1 provides indirect data on that point, illustrating where the program counter was when the signal occurred. There is a strong peak at 0—in over one quarter of all test cases, when the program was stopped it was on the first byte of the randomized attack code, and the fraction of attacks falls off rapidly at increasing offsets.

Another caveat in this test is that we don't know exactly how many instructions were executed before the signal occurred. Random control transfers occur frequently, so the location of a signal does not correlate directly with number of instructions executed. As seen in Figure 1, for example, a cumulative total of about 6% of the signaling cases occurred at addresses below the starting point of the attack.

Using RISE itself, we can address the question of how many instructions are executed, because it is easy for an emulator to count how many instructions it has emulated. However, it is much more expensive to collect data this way. Table 3 gives results for a few concrete data points. We show data on three real attacks against vulnerable programs, with an average of under five instructions being executed before the attacked program is stopped (column 4). Column 3 indicates how many attack instances we ran (each with a different random seed for RISE) to compute the average. As column 5 shows, most attack instances were stopped by an attempt to execute a non-existent opcode. In addition, we ran the two synthetic attacks (described earlier) one hundred times each (with a new seed each time) and discovered that neither attack ever executed successfully. On average, each synthetic attack instance executed 2.35 bytes of instructions before process death.

Within the RISE approach, one could avoid the problem of accidentally viable code by mapping to a larger instruction set. The size could be tuned to reflect the desired percentage of incorrect unscramblings that will likely lead immediately to an illegal instruction.

Attack Name	**Application**	**No. of attacks**	**Avg. no. of insns.**	**Illegal insn.**
Named NXT Resource Record Overflow	Bind 8.2.1-7	33	2.84	84%
rpc.statd format string	nfs-utils 0.1.6-2	25	4.13	80%
Samba trans2 exploit	smbd 2.2.1a	81	3.13	73%

Table 3: Survival time in executed instructions for attack code in real applications running under RISE. Column 4 gives the average number of instructions executed before failure, and column 5 summarizes the percentage of runs failing because of illegal instructions.

3.3 Performance

There is a significant cost introduced by the memory checking engine of Valgrind. However, RISE adds only a modest performance penalty beyond that. In terms of execution time, a RISE-protected program executes about 5% more slowly than the same program running under Valgrind; we believe much of that slowdown is due to the relatively high cost of the `mprotect` system calls used to control modifications of the trace cache. In terms of space, significant impacts come from the scrambling information and the private copies of shared libraries, each of which requires about as much space as the protected code.

We have been able to RISE-protect every one of the services used in the experiments (httpd, named, cvs pserver, smbd, sshd, rpc.statd, sendmail, wuftpd) on a 200 MHz Pentium computer with 128 MB RAM, and run it with reasonable response time. This is a far smaller and slower machine than any modern x86-based server system, which gives us confidence that the memory expense does not make the scheme impractical and would be a reasonable tradeoff for increased security.

4. RELATED WORK

Our randomization technique is an example of *automated diversity*, an idea that has long been used in software engineering to improve fault tolerance [9, 34, 10], and more recently has been proposed as a method for improving security [17, 24, 19]. An approach similar to RISE, but focusing on a whole-system emulator, is proposed in [27]. Several other (nondiversifying) approaches have been developed for protecting against stack-smashing attacks, a method of code injection [40, 20, 21, 25]. Instruction-set randomization is also related to hardware encryption methods as protection against piracy and eavesdropping for specialized applications [12, 13, 22] and general purpose systems [29, 5].

4.1 Automated diversity

Diversity in software engineering is quite different from diversity for security. In software engineering, the basic idea is to generate multiple independent solutions to a problem (e.g., multiple versions of a software program) with the hope that they will fail independently, thus greatly improving the chances that some solution out of the collection will perform correctly in every circumstance. The different solutions may or may not be produced manually, and the number of solutions is typically quite small, around ten.

Diversity in security is introduced for a different reason. Here, the goal is to reduce the risk of widely replicated attacks, by forcing the attacker to redesign the attack each time it is applied. For example, in the case of a buffer overflow attack, the goal is to force the attacker to rewrite the attack code for each new computer that

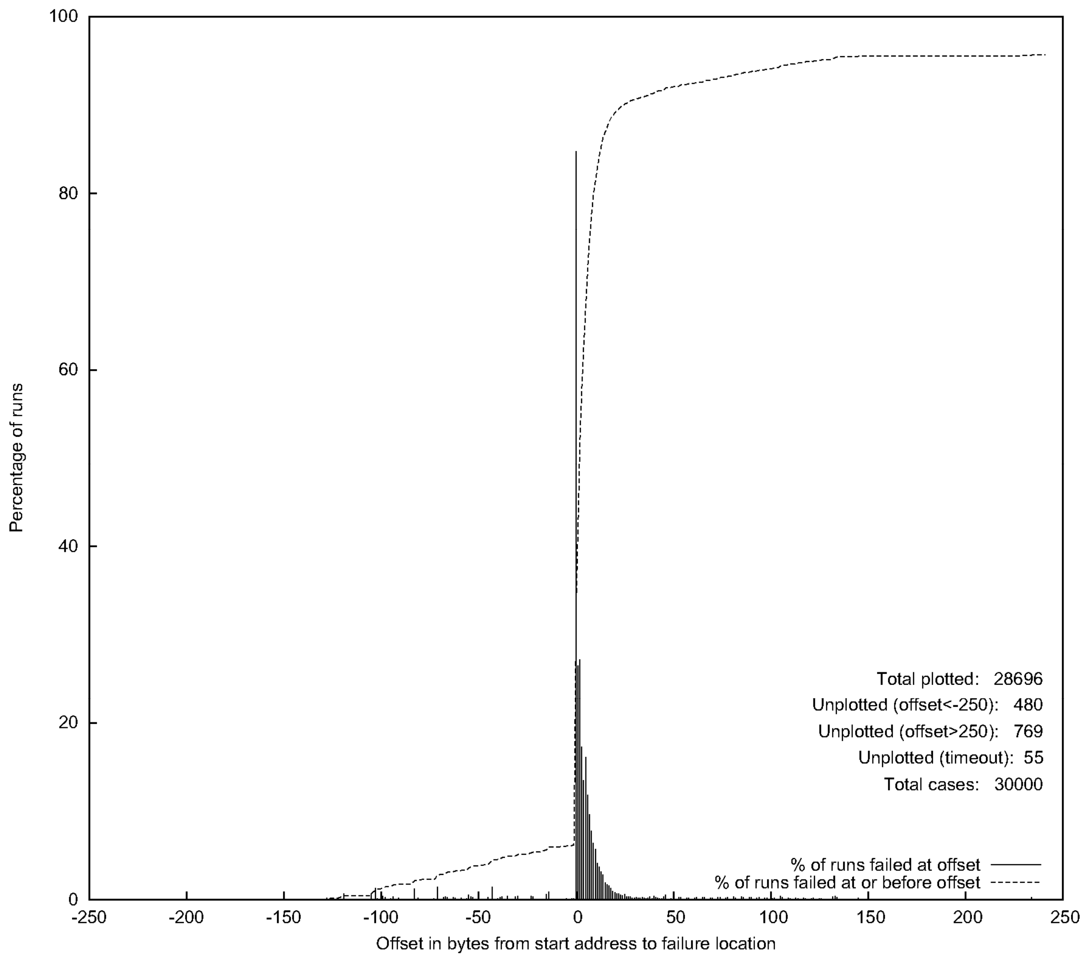

Figure 1: Distribution of locations at which the synthetic attack was stopped.

is attacked. Here the number of different diverse solutions is very high, potentially equal to the total number of program copies for any given program. Manual methods are infeasible here, and the diversity must be produced auomatically.

Cowan et al. introduced a classification of diversity methods applied to security (called 'security adaptations') which classifies adaptations based on what is being adapted, either the interface or the implementation [19]. Interface adaptations modify code layout or access controls to interfaces, without changing the underlying implementation to which the interface gives access. Implementation adaptations, on the other hand, do modify the underlying implementation of some portion of the system to make it resistant to attacks. RISE can be viewed as an interface randomization at the machine code level.

Earlier work in automated diversity for security has experimented with diversifying data layouts [17, 33], file systems [19], and system-call interfaces [16]. In addition, several projects address the code-injection threat model directly, and we describe those projects briefly.

In 1997, Forrest et al. presented a general view of the possibilities of diversity for security [24], introducing the idea of deliberately diversifying data layouts as well as code, and demonstrated an example of diversification that randomly padded stack frames so that exact return address locations would be less predictable, making it harder for an attacker to locate the return address and other key stack offsets. Developers of buffer overflow attacks have developed a variety of workarounds—such as 'ramps' and 'landing zones' of no-ops and multiple return addresses—aimed at coping with variations across different versions or different compilations of the vulnerable software. Deliberate diversification via random stack padding coerces an attacker to use such generalization techniques; it also necessitates larger attack codes in proportion to the size range of random padding employed.

The StackGuard system [20] provides a counter-defense against landing zones and similar attack techniques by interposing a hard-to-guess 'canary word' before the return address, the value of which is checked before the function returns. An attempt to overwrite the return address via linear stack smashing will almost surely change the canary value and thus be detected.

4.2 Enforcing security with optimizing interpreters

It has been noted that the current trend in binary-to-binary optimizing interpreters could be used for more detailed inspection of executing code, because every control transfer is detected during the interpretation process. Kiriansky et al. [28] proposed a method called 'code shepherding' in which various policies are defined to govern allowable control transfers. Two of those types of policies are relevant to the RISE approach.

Code origins policies grant differential access based on the source of the code. When it is possible to establish if the instruction to be executed came from a disk binary (modified or unmodified) or from dynamically generated code (original or modified after generation), policy decisions can be made based on that origin information. In our model, we are implicitly implementing a code-origin policy, in that only unmodified code from disk is allowed to execute. An advantage of the RISE approach is that the origin check cannot be avoided—only properly sourced code is mapped into the private instruction set so it executes successfully. Currently, the only exception we have to the disk-origin code policy is the code deposited in the stack by signals, which is handled specially by Valgrind. Also relevant are *restricted control transfers* in which a transfer is allowed or disallowed according to its source, destination, and type. Although we use a restricted version of this policy to allow signal code on the stack, in other cases we rely on the RISE 'language barrier' to ensure that injected code will fail.

4.3 Other defenses against buffer overflows

In addition to the stack-frame padding and canary methods [40, 20] described earlier, several other solutions have been proposed to deal specifically with buffer overflows [21]. These solutions employ compiler extensions [20, 24], hardware characteristics [25, 32], kernel modifications [37, 32], library modifications [3], or static analysis [41] to prevent and detect exploitation of buffer overflow vulnerabilities.

RISE shares many of the advantages of non-executable stack and heap techniques, including the ability to randomize ordinary executable files and no special compilation requirements however. Our approach differs, however, from non-executable stacks and heaps in important ways. First, most non-executable stack/heap systems (such as PaX [32]) are applied systemwide, while RISE can be selectively employed on a per-process basis. This distinction becomes important, for example, when we consider Java Virtual Machines, where a runtime compilation process generates code, places it on the heap, then later jumps to it. In a system with a traditional non-executable heap, JVMs cannot run at all. In RISE, the JVM process can simply be run outside of RISE without compromising the security of other running processes. Second, enabling non-executable stack/heap protection on a system often requires additional hardware or operating system modification. RISE runs as a user-level application and requires no special hardware or OS changes. RISE is capable of running on any binary-to-binary translator, and so can run on any system with such software.

4.4 Hardware encryption

Because RISE uses runtime code scrambling to improve security, it resembles some hardware-based code encryption schemes. Hardware components to allow decryption of code and/or data on-the-fly have been proposed since the late 70's [12, 13] and implemented as microcontrollers for custom systems (for example the DS5002FP microcontroller [22]). The two main objectives of these cryptoprocessors are to protect code from piracy and data from in-chip eavesdropping. An early proposal for the use of hardware encryption in general purpose systems was presented by Kuhn for a very high threat level where the encryption and decryption was performed at the level of cache lines [29]. This proposal still adheres to the model of protecting licensed software from users, and not users from intruders, so there is no analysis of how to deal with shared libraries or how to encrypt (if desired) existing open applications. A more extensive proposal was included as part of TCPA/TCG [5]. Although the published TCPA/TCG specifications provide for encrypted code in memory, which is decrypted on the fly, TCPA/TCG is designed as a much larger authentication and verification scheme and has raised controversies about Digital Rights Management (DRM) and end-users losing control of their systems ([7],[8]). RISE contains none of the machinery found in TCPA/TCG for supporting DRM. On the contrary, RISE is designed to maintain control locally to protect the user from injected code.

5. DISCUSSION AND CONCLUSIONS

In this paper we introduced the concept of a randomized instruction set emulator as a defense against binary code injection attacks. We demonstrated the feasibility and utility of this concept with a proof-of-concept implementation based on Valgrind. Our implementation successfully scrambles binary code at load time, unscrambles it instruction-by-instruction during instruction fetch, and executes the unscrambled code correctly. The implementation was successfully tested on several code-injection attacks, some real and some synthesized to exhibit common injection techniques.

Although Valgrind has some limitations, discussed in Section 2, we are optimistic that improved designs and implementations of "randomized machines" would vastly increase performance and reduce resource requirements, potentially expanding the range of attacks the approach can mitigate. In the current implementation, aside from performance issues, there is a potential concern about the dense packing of legal x86 instructions in the space of all possible byte patterns. A random scrambling of bits is likely to produce a different legal instruction. Doubling the size of the instruction encoding would enormously reduce the risk of a processor successfully executing a long enough sequence of undescrambled instructions to do damage. Although our preliminary analysis shows that this risk is low even with the current implementation, we believe that emerging 'soft-hardware' architectures such as Crusoe will make it possible to reduce the risk even further.

A valid concern when evaluating RISE's security is its susceptibility to key discovery, as an attacker with the appropriate scrambling information could inject scrambled code which will be accepted by the emulator. We believe that RISE is highly resistant to this class of attacks.

RISE is resilient against brute force attacks because the attacker's work is exponential in the shortest code sequence that will make an externally detectable difference if it is unscrambled properly. We can be optimistic because most x86 attack codes are at least dozens of bytes long, but if a software flaw existed that was exploitable with, say, a single one-byte opcode, then RISE would be vulnerable, although the process of guessing even a one-byte representation would cause system crashes easily detectable by an administrator.

An alternative path for an attacker is to try to dump arbitrary address ranges of the process into the network, and recover the key from the downloaded information. The download could be part of the key itself (stored in the process address space), scrambled code, or unscrambled data. Unscrambled data does not give the attacker any information about the key. Even if the attacker obtains scrambled code or pieces of the key (they are equivalent because we can assume that the attacker has knowledge of the program binary), using the stolen key piece might not be feasible. If the key is created eagerly, with a key for every possible address in the program, past or future, then the attacker would still need to know where the attack code is going to be written in process space to be able to use that information. However, in our implementation, where keys are created lazily for code loaded from disk, the key for the addresses targeted by the attack might not exist, and therefore might not be discoverable. The keys that do exist are for addresses that are usually not used in code injection attacks because they are write protected. In summary, it would be extremely difficult to discover or use a particular encoding during the lifetime of a process.

An attraction of RISE, compared to an approach such as code shepherding, is that injected code is stopped by an inherent property of the system, without requiring any explicit or manually defined checks before execution. Although divorcing policy from mechanism (as in code shepherding) is a valid design principle in general, it is very easy to make mistakes in defining security policies, and a mechanism that inherently enforces a correct policy is desirable.

An essential requirement for using RISE for improving security is that the distinction between code and data must be carefully maintained. The discovery that code and data can be systematically interchanged was a key advance in early computer design, and that dual interpretation of bits as both numbers and commands is inherent to programmable computing. However, all that flexibility and power turn into security risks if we cannot control how and when data become interpreted as code. Code injection attacks provide a compelling example, as the easiest way to inject code into a binary is by disguising it as data, e.g., as arguments to functions in a victim program.

Fortunately, code and data are typically used in very different ways, so advances in computer architecture intended solely to improve performance, such as separate instruction caches and data caches, also have helped enforce good hygiene in distinguishing machine code from data, helping make the RISE approach feasible. At the same time, of course, the rise of mobile code, such as Javascript in web pages and macros embedded in word processing documents, tends to blur the code/data distinction and create new risks.

Although our paper illustrates the idea of randomizing instruction sets at the machine code level, the basic concept could be applied wherever it is possible to (1) distinguish code from data, (2) identify all sources of trusted code, and (3) introduce hidden diversity into all and only the trusted code. A RISE for protecting `printf` format strings, for example, might rely on compile-time detection of legitimate format strings, which might either be randomized upon detection, or flagged by the compiler for randomization sometime closer to runtime. Certainly, it is essential that a running program interact with external information, at some point, or no externally useful computation can be performed. However, as the recent SQL attacks illustrate [26], it is increasingly dangerous to express running programs in externally known languages. Randomized instruction set emulators are one step towards reducing that risk.

As the complexity of systems grows, and 100% provable overall system security seems an ever more distant goal, the principle of diversity suggests that having a variety of defensive techniques based on different mechanisms with different properties stands to provide increased robustness, even if the techniques address partially or completely overlapping threats. Exploiting the idea that it's hard to get much done when you don't know the language, RISE is another technique in the defender's arsenal against binary code injection attacks.

Acknowledgments

The authors gratefully acknowledge the partial support of the National Science Foundation (grants ANIR-9986555, CCR-0219587, CCR-0085792, CCR-0311686, EIA-0218262, and EIA-0238027), the Office of Naval Research (grant N00014-99-1-0417), Defense Advanced Research Projects Agency (grants AGR F30602-00-2-0584 and F30602-02-1-0146), Sandia National Laboratories, Hewlett-Packard, Microsoft Research, Intel Corporation, and the Santa Fe Institute.

6. REFERENCES

[1] CORE Security Technologies. In *http://www1.corest.com/home/home.php*.

[2] CVS Directory Request Double Free Heap Corruption Vulnerability. In *http://www.securityfocus.com/bid/6650*.

[3] libsafe - Detect and handle buffer overflow attacks. In *http://www.gnu.org/directory/ security/net/libsafe.html*.

[4] MySQL COM_CHANGE_USER Password Length Account Compromise Vulnerability. In *http://www.securityfocus.com/bid/6373*.

[5] TCPA Trusted Computing Platform Alliance. In *http://www.trustedcomputing.org/home*.

[6] Aleph One. Smashing the stack for fun and profit. *Phrack*, 49(7), Nov. 1996.

[7] R. Anderson. 'Trusted Computing' and competition policy - issues for computing professionals. *Upgrade*, IV(3):35–41, June 2003.

[8] W. A. Arbaugh. Improving the TCPA specification. *IEEE Computer*, 35(8):77–79, August 2002.

[9] A. Avizienis. The Methodology of N-Version Programming. In M. Lyu, editor, *Software Fault Tolerance*, pages 23–46. John Wiley & Sons Ltd., 1995.

[10] A. Avizienis and L. Chen. On the implementation of N-Version programming for software fault tolerance during execution. In *Proceedings of IEEE COMPSAC 77*, pages 149–155, Nov. 1977.

[11] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation*, pages 1–12, Vancouver, British Columbia, Canada, 2000. ACM Press.

[12] R. M. Best. Microprocessor for executing enciphered programs, U.S. Patent No. 4 168 396, September 18 1979.

[13] R. M. Best. Preventing software piracy with crypto-microprocessors. In *Proceedings of the IEEE Spring COMPCON '80*, pages 466–469, San Francisco, California, Feb. 1980.

[14] S. Bhatkar, D. DuVarney, and R. Sekar. Address obfuscation: An approach to combat buffer overflows, format-string attacks and more. In *12th Usenix Security Symposium*, Aug. 2003.

[15] D. Bruening, S. Amarasinghe, and E. Duesterwald. Design and implementation of a dynamic optimization framework for Windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, Dec. 2001.

[16] M. Chew and D. Song. Mitigating Buffer Overflows by Operating System Randomization. Technical Report CMU-CS-02-197, Department of Computer Science, Carnegie Mellon University, Dec. 2002.

[17] F. Cohen. Operating System Protection through Program Evolution. *Computers and Security*, 12(6):565–584, Oct. 1993.

[18] C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman. Format guard: Automatic protection from `printf` format string vulnerabilities. In *Proceedings of the 2001 USENIX Security Symposium*, Washington DC, August 2001.

[19] C. Cowan, H. Hinton, C. Pu, and J. Walpole. A Cracker Patch Choice: An Analysis of Post Hoc Security Techniques. In *National Information Systems Security Conference (NISSC)*, Baltimore MD, October 16-19 2000.

[20] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Automatic Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, Texas, Jan. 1998.

[21] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, pages 119–129, Jan. 2000.

[22] Dallas Semiconductor. DS5002FP secure microprocessor chip. http://pdfserv.maxim-ic.com/en/ds/DS5002FP.pdf.

[23] P. Fayolle and V. Glaume. A buffer overflow study, attacks & defenses. In *http://www.enseirb.fr/~glaume/indexen.html.*

[24] S. Forrest, A. Somayaji, and D. Ackley. Building Diverse Computer Systems. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, pages 67–72, 1997.

[25] M. Frantzen and M. Shuey. Stackghost: Hardware facilitated stack protection. In *Proceedings of the 10th USENIX Security Symposium*, Washington D.C., August 2001.

[26] M. Harper. SQL injection attacks - are you safe? In *Sitepoint, http://www.sitepoint.com/article/794*, June 17 2002.

[27] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *www.cs.columbia.edu/~gskc/publications/isaRandomization.pdf.*

[28] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution Via Program Sheperding. In *Proceeding of the 11th USENIX Security Symposium*, San Francisco, California, August 2002.

[29] M. Kuhn. The TrustNo 1 cryptoprocessor concept. Technical Report CS555 Report, Purdue University, April 04 1997.

[30] Nergal. The advanced return-into-lib(c) exploits. *Phrack*, 58(4), Dec. 2001.

[31] T. Newsham. Format string attacks. In *http://www.securityfocus.com/archive/1/81565*, September 9 2000.

[32] PaX team. Non executable data pages. In *http://pageexec.virtualave.net/ pageexec.txt*, 2002.

[33] C. Pu, A. Black, C. Cowan, and J. Walpole. A specialization toolkit to increase the diversity of operating systems. In *Proceedings of the 1996 ICMAS Workshop on Immunity-Based Systems*, Nara, Japan, December 1996.

[34] B. Randell. System Structure for Software Fault Tolerance. *IEEE Transactions in Software Engineering*, 1(2):220–232, 1975.

[35] B. Schneier. *Applied Cryptography*. John Wiley & Sons, 1996.

[36] J. Seward. Valgrind, an open-source memory debugger for x86-GNU/Linux. In *http://developer.kde.org/˜sewardj/*, 2002.

[37] Solar Designer. Non-executable user stack. In *http://www.openwall.com/linux*.

[38] Tool Interface Standards Committee. *Executable and Linking Format (ELF)*, May 1995.

[39] T. Tso. random.c A strong random number generator. In *http://www.linuxsecurity.com/feature_stories/random.c*.

[40] Vendicator. StackShield: A stack smashing technique protection tool for Linux. In *http://angelfire.com/sk/stackshield*.

[41] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A First Step towards Automated Detection of Buffer Overrun Vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, February 2000.